The NMI Build & Test Laboratory: Continuous Integration Framework for Distributed Computing Software

Andrew Pavlo, Peter Couvares, Rebekah Gietzel, Anatoly Karp, Ian D. Alderman, and Miron Livny – University of Wisconsin-Madison
Charles Bacon – Argonne National Laboratory

ABSTRACT

We present a framework for building and testing software in a heterogeneous, multi-user, distributed computing environment. Unlike other systems for automated builds and tests, our framework is not tied to a specific developer tool, revision control system, or testing framework, and allows access to computing resources across administrative boundaries. Users define complex software building procedures for multiple platforms with simple semantics. The system balances the need to continually integrate software changes while still providing on-demand access for developers. Our key contributions in this paper are: (1) the development of design principles for distributed build-and-test systems, (2) a description of an implemented system that satisfies those principles, and (3) case studies on how this system is used in practice at two sites where large, multi-component systems are built and tested.

Introduction

Frequently building and testing software yields many benefits [10, 14, 17]. This process, known as *continuous integration*, allows developers to recognize and respond to problems in their applications as they are introduced, rather than be inundated with software bugs only when a production release is needed [2]. If the time span between the last successful build of an application and latest broken version is short, it is easier to isolate the source code modifications that caused the application's compilation or testing to fail [17]. It is important to fix these software bugs early in the development process, as the cost of the fix has been shown to be proportional to the age of the bug [3].

We developed the NMI Build & Test framework to facilitate automatic builds and tests of distributed computing software in a distributed computing environment. It is part of the NSF Middleware Initiative (NMI), whose mission is to develop an integrated national middleware infrastructure in support of science and engineering applications. In these types of problem domains, build-and-test facilities may be comprised of a few computing resources in a single location, or a large, heterogeneous collection of machines in different geographical and administrative domains. Our system abstracts the build-and-test procedures from the underlying technology needed to execute these procedures on multiple resources. The logically distinct steps to build or test each application may be encapsulated in separate, fully automated tasks in the framework without restricting users to any specific development tool or testing system. Thus, developers can migrate their existing build-and-test procedures easily without compromising the procedures of other applications using the framework.

To build and test any application, users explicitly define the execution workflow of build-and-test procedures, along with any external software dependencies and target platforms, using a lightweight declarative syntax. The NMI Build & Test software stores this information in a central repository to ensure every build or test is reproducible. When a build or test *routine* is submitted to the framework, the procedures are dynamically deployed to the appropriate computing resources for execution. Users can view the status of their routines as they execute on build-and-test resources. The framework captures any artifacts produced during this execution and automatically transfers them to a central repository. Authorized users can pause or remove their routines from the framework at any time.

We implement the NMI framework as a lightweight software layer that runs on top of the Condor high-throughput distributed batch computing system [15, 25]. Leveraging a feature-rich batch system like Condor provides our framework with the fault-tolerance, scheduling policies, accounting, and security it requires. The NMI Build & Test software is not installed persistently on all available computing resources; it is deployed dynamically by the batch system at runtime.

The framework and software that we present here are just one component of the NMI Build & Test Laboratory at the University of Wisconsin-Madison's Department of Computer Sciences. The Laboratory also provides maintenance and administration for a diverse collection of resources. It is used as the primary build-and-test environment for the Globus Toolkit [8] and the Condor batch system [25], as well as other products. Managing such a production facility

presents certain specific system administration problems, such as maintaining a standard set of software libraries across multiple platforms and coping with the large amount of data produced daily by our users.

In this document, we discuss not only the design and architecture of the NMI framework and software, but also the tools and practices we developed for managing a heterogeneous build-and-test facility on which it is deployed.

Related Work

Continuous integration and automated build-and-test systems are used by many large software projects [14]. The benefits of these systems are most often reported in discussions of agile software development [2, 12, 22]. Research literature on the general design of such systems, however, is limited.

There are numerous commercial and open source continuous integration and automated build systems available [13]. Almost all provide the basic functionality of managing build and test execution on one or more computing resources. Three popular systems are the Mozilla Project's Tinderbox system [20], the Apache Software Foundation's Maven [16], and the CruiseControl toolkit [6]. The Tinderbox system requires autonomous agents on build machines to continuously retrieve source code from a repository, compile the application, and send status reports back to a central server. This is different from the approach taken by Maven and CruiseControl, where a central manager pushes builds and tests to computing resources and then retrieves the results when they are complete.

Many systems make important assumptions about the scope and complexity of the computing environment in which they are deployed. For example, some require that all build-and-test resources be dedicated or that all users have equal access to them. Other systems assume that prerequisite software is predictably installed and configured by the system administrator on all machines in the pool. Users must hard-code paths to these external dependencies in their build-and-test scripts, making it difficult to reproduce past routines on platforms that have been patched or updated. Although these constraints may be appropriate for smaller projects with few users, they are not realistic for larger organizations with diverse administrative controls or projects involving developers located throughout the world.

Other systems offer more flexibility and control of the build-and-test execution environment. The ElectricCloud commercial distributed build system re-factors an application's Makefiles into parallel workloads executed on dedicated clusters [19]. A central manager synchronizes the system clocks for the pool to help ensure that a build script's time stamp-based dependencies work correctly. Another full-featured commercial offering is the BuildForge continuous integration system [7]. It uses an integrated batch system to provide rudimentary opportunistic computing capabilities and resource controls based on user and group policies.

These systems seldom address the many problems inherent in managing workloads in a distributed environment, however. For example, a system must ensure that a running build or test can be cancelled and completely removed from a machine. This is often not an easy accomplishment; thorough testing of an application often requires additional services, such as a database server, to be launched along with the application and testing scripts may leave a myriad of files scattered about the local disk.

Motivation

In a distributed computing environment, a build-and-test system cannot assume central administrative control, or that its resources are dedicated or reliable. Therefore, we need a system that can safely execute routines on resources outside of one local administrative domain. This also means that our system cannot assume that each computing resource is installed with software needed by the framework, or configured identically.

Because of the arbitrary nature of how routines execute in this environment, non-dedicated remote computing resources are often less reliable than local build-and-test machines. But even large pools of dedicated resources begin to resemble opportunistic pools as their capacity increases, since hardware failure is inevitable. A routine may be evicted from its execution site at any time. We need a system that can restart a routine on another resource and only execute tasks that have not already completed. The build-and-test framework should also ensure that routines are never "lost" in the system when failures occur.

Lastly, we need a mechanism for describing the capabilities, constraints, and properties of heterogeneous resources in the pool. With this information, a system ensures that each build-and-test routine is matched with a machine providing the correct execution environment: a user may require that their build routines only execute on a computing resource with a specific software configuration. The system needs to schedule the routine on an available matching machine or defer execution until a machine satisfying the user's requirements becomes available. If a satisfying resource does not exist, the system needs to notify the user that their requirements cannot be met.

Design Principles

The NMI framework was designed in response to our experiences developing distributed computing software. Our first implementation was created to help merge the build-and-test procedures of two large software projects into a unified environment where they could share a single pool of computing resources and

be packaged into a single grid software distribution. Both projects already had different established practices for building and testing their applications using a menagerie of custom scripts and build tools. Thus, our goal was to develop a unified framework incorporating these application-specific scripts and processes.

We developed a set of design principles for distributed build-and-test systems to solve the problems that we encountered in this merging process. We incorporated these principles into our implementation of the NMI Build & Test system. They are applicable to other continuous integration frameworks, both large and small.

Tool Independent

The framework should not require a software project to use a particular set of development or testing tools. If the build-and-test procedures for an application are encapsulated in well-defined building blocks, then a clear separation of the blocks and the tools used to manipulate them permits diversity. In our system, users are provided with a general interface to the framework that is compatible with arbitrary build-and-test utilities. The abstraction afforded by this interface ensures that new application-specific scripts can be integrated without requiring modifications to, and thereby affecting the stability of, the framework or other applications.

Lightweight

The software should be small and portable. This approach has three advantages: (1) it is easier for system administrators to add new resources to a build-and-test pool, (2) users are able to access resources outside of their local administrative domain where they may be prohibited from installing persistent software, and (3) framework software upgrades are easier as only the submission hosts need to be updated.

The NMI Build & Test framework uses existing, proven tools to solve many difficult problems in automating a distributed computing workflow. Because it is designed to be lightweight, it is able to run on top of the Condor batch system and take advantage of the workload management and distributed computing features Condor offers. The NMI software only needs to be installed on submission hosts, where it is deployed dynamically to computing resources. By this we mean that a subset of the framework software is transferred to build-and-test resources and automatically deployed at runtime.

Explicit, Well-Controlled Environments

All external software dependencies and resource requirements for each routine must be explicitly defined. This helps to ensure a predictable, reproducible, and reliable execution environment for a build or test, even on unpredictable and unreliable computing resources.

When a routine's procedures are sent to a build-and-test resource for execution in the NMI system, the framework creates and isolates the proper execution environment on demand. The framework ensures that only the software required by the routine is available at run time. This may be accomplished in two ways: (1) the developer must declare all the external software that their application requires other than what exists in the default vendor installation of the operating system, or (2) the developer may use the framework interface to automatically retrieve, configure, and temporarily install external software in their routine's runtime environment.

Central Results Repository

A build-and-test system should capture all information and data generated by routines and store it in a central repository. It is important that system allows users to easily retrieve the latest version of applications and view the state of their builds and tests [10]. The repository maintains routine's provenance information and historical data, which can be used for statistical analysis of builds and tests.

The NMI framework stores the execution results, log files, and output created by routines, as well as all input data, environment information, and dependencies needed to reproduce the build or test. While a routine executes, the NMI Build & Test software continuously updates the central repository with the results of each procedure; users do not need to wait for a routine to finish before viewing its results. Any output files produced by builds or tests are automatically transferred back to the central repository.

Fault Tolerance

The framework must be resilient to errors and faults from arbitrary components in the system. This allows builds and tests to continue to execute even when a database server goes down or network connectivity is severed. If the NMI Build & Test software deployed on a computing resource is unable to communicate with the submission host, the routine executing on that resource continues unperturbed. When the submission host is available again, all queued information is sent back; routines never stop making forward progress because the framework was unable to store the results.

The framework also uses leases to track an active routine in the system. If the framework software is unable to communicate with a resource executing a routine, the routine is not restarted on another machine until its lease expires. Thus, there are never duplicate routines executing at the same time.

Platform-Independent vs. Specific

For multi-platform applications, users should be able to define platform-independent tasks that are only executed once per routine submission. This improves the overall throughput of a build-and-test pool. For

example, an application's documentation only needs to be generated once for all platforms.

Build/Test Separation

The output of a successful build can be used as the input to another build, or to a future test. Thus, users are be able to break distinct operations into smaller steps and decouple build products from testing targets. As described above, the framework archives the results of every build and test. When these cached results are needed by another routine as an input, the framework automatically transfers the results and deploys it on the computing resource at run time.

NMI Software

We developed the NMI Build & Test Laboratory's continuous integration framework software based on the design principles described in the previous section. The primary focus of our framework is to enable software to be built and tested in a distributed batch computing environment. Our software provides a command-line execution mechanism that can be triggered by any arbitrary entity, such as the UNIX cron daemon or a source code repository monitor, or by users when they need to quickly build their software before committing changes [10]. We believe that it is important for the framework to accommodate diverse projects' existing development practices, rather than force the adoption of a small set of software.

The NMI framework allows users to submit builds and tests for an application on multiple resources from a single location. We use a batch system to provide all the network and workload management functionality. The batch system is installed on every machine in a build-and-test pool, but the NMI software is only installed on the submission hosts. The framework stores all information about executing routines in a central database. The output from routines is returned to the submission hosts, which can store them on either a shared network storage system or an independent file system.

A build-and-test routine is composed of a set of *glue scripts* and a *specification file* containing information about how an application is built or tested. The glue scripts are user-provided, application-specific tasks that automate parts of the build-and-test process. These scripts together contain the steps needed to configure, compile, or deploy an application inside of the framework. The specification file tells the framework when to execute these glue scripts, which platforms to execute them on, how to retrieve input data, and what external software dependencies exist.

Workflow Stages

The execution steps of a framework submission are divided into four stages: fetch, pre-processing, platform, and post-processing (see Figure 1). The tasks in the pre- and post-processing stages can be distributed on multiple machines to balance the workload. A routine's results and output are automatically transferred to and stored on the machine that it was submitted from.

Fetch: In this stage, the framework retrieves all the input data needed to build or test an application. Instead of writing custom scripts, users declare where and how files are retrieved using templates provided by the framework. Input data may come from multiple sources, including source code repositories (cvs, svn), file servers (http, ftp), and the output results from previous builds. Thus, input templates document the provenance of all inputs and help ensure the repeatability of routines.

Pre-processing: This optional stage prepares the build-and-test routine for execution on computing resources. These tasks are often used to process the input data collected in the previous stage. The platform-independent tasks execute

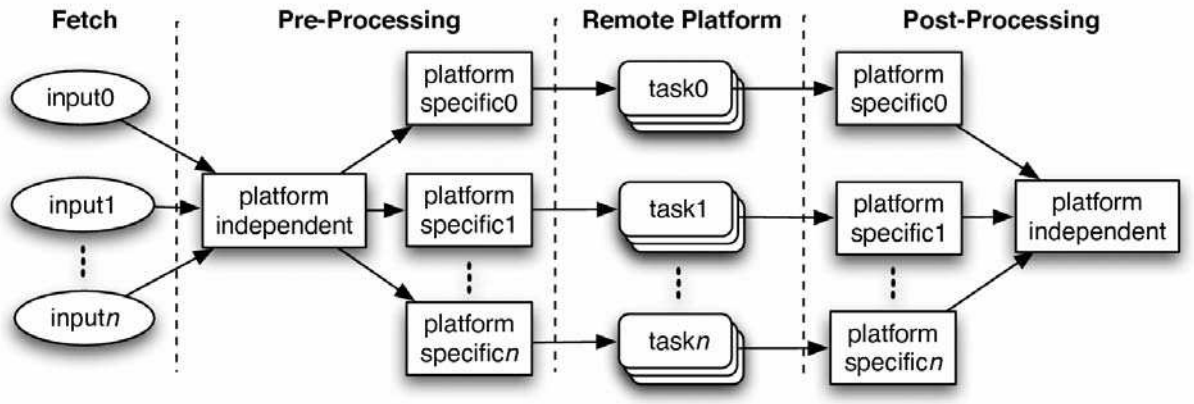

Figure 1: Workflow Stages – The steps to build or test an application in the NMI framework are divided into four stages. The fetch stage is executed on the machine that the user submitted the routine. The pre- and post-processing stages execute on any resource. The remote platform tasks each execute on the appropriate platform.

first and may modify the input data for all platforms. The framework then makes separate copies of the potentially modified input data for each platform and executes the platform-specific tasks. Any modifications made to the input data by the platform-specific tasks are only reflected in that platform's copy.

Remote platform: After the input data is retrieved and processed, the framework submits one job for each target platform to the batch system. These jobs spawn the remote platform tasks to build or test an application on an appropriate compute resource. The NMI framework tells the batch system which input files to transfer to the resource along with a copy of the remote NMI framework software and the platform task glue scripts. Before these scripts begin to execute, the NMI software prepares the working directory for the routine and binds the execution environment paths to the local configuration of the machine. When each task finishes, any output produced can be sent back to the submission host for storage.

Post-processing: This stage contains tasks that process the output data produced by routines executing on build-and-test resources. As the platform tasks complete for each platform, the framework executes the platform-specific scripts for the corresponding set of results. Once these tasks are completed for all the platforms, the platform-independent scripts are then executed.

Workflow Manager

Using a distributed batch system to coordinate the execution of jobs running on the build-and-test machines provides the NMI framework with the robustness and reliability needed in a distributed computing environment.

We use the Directed Acyclic Graph Manager (DAGMan) to automate and control jobs submitted to the batch system by the NMI Build & Test software [5, 25]. DAGMan is a meta-scheduler service for executing multiple jobs in a batch system with dependencies in a declarative form; it monitors and schedules the jobs in a workflow. These workflows are expressed as directed graphs where each node of the graph denotes an atomic task and the directed edge indicates a dependency relationship between two adjacent nodes.

When a routine is submitted to the framework, its specification file is transformed into an execution graph. A single instance of DAGMan with this graph as its input is submitted to the batch system. DAGMan can then submit new jobs to the batch system using a standard application interface. As each of its spawned jobs complete, DAGMan is notified and can deploy additional jobs based on the dependencies in the graph.

DAGMan also provides the NMI Build & Test software with fault-tolerance. It is able to checkpoint a workflow much like a batch system is able to checkpoint a job. If the batch system fails and must be restarted, the workflow is restarted automatically and DAGMan only executes tasks that have not already completed.

Glue Scripts

A routine's glue scripts contain the procedures needed to build or test an application using the NMI framework. These scripts automate the typical human-operated steps so that builds and tests require no human intervention. Build glue scripts typically include configure, compile, and package steps. Test glue scripts can deploy additional services or sub-systems at runtime for thorough testing and can use any testing harness or framework.

The framework provides a glue script with information about the progress of its routine through predefined environment variables. Thus, the scripts can

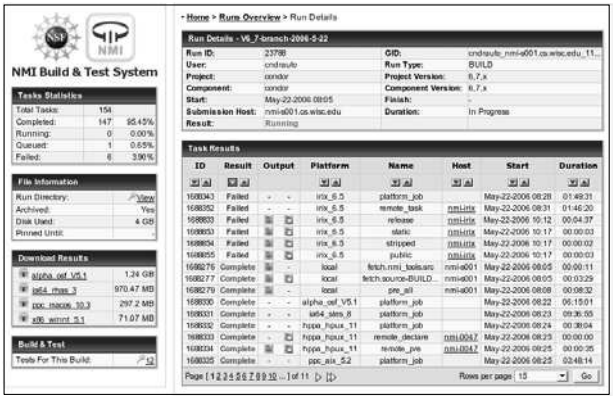

Figure 2(a): Routine status

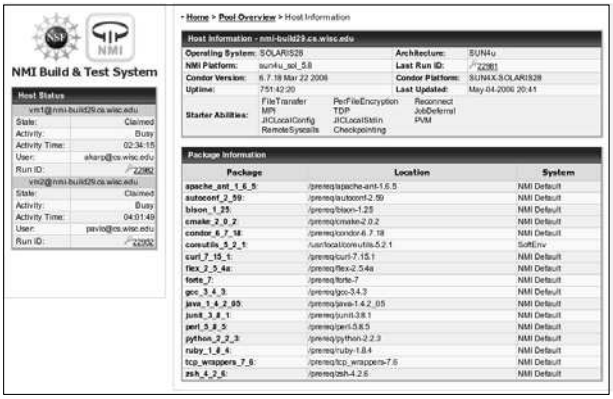

Figure 2(b): Computing resource information

Figure 2: NMI Framework Web Interface – The NMI Build & Test software provides a web client for users to view information about their build-and-test system. The screenshot in Figure 2(a) shows status information about a routine submitted to the framework; users can monitor the progress of tasks, download output files, and view log files. The screenshot in Figure 2(b) shows the capabilities of a machine, lists all prerequisite software installed, and provides information about the routines currently executing on it.

control a routine's execution workflow while they are running on a build-and-test resource. For example, a build glue script might halt execution if a dependency failed to compile in a previous step. Optionally, a test glue script may choose to continue even if the previous test case failed.

Application Interfaces

The NMI framework provides a standard interface for submitting and managing routines in a build-and-test system. This interface can easily be augmented by other clients or notification paradigms. For example, our framework distribution includes a web interface that provides an up-to-date overview of the system (Figure 2).

Batch System

We designed the NMI framework to run on top of the Condor high-throughput distributed computing batch system [15, 25]. When a user submits a build-and-test routine, the framework software deploys a single DAGMan job into Condor (Figure 3). This DAGMan job then spawns multiple Condor jobs for each platform targeted by the routine. Condor ensures that these jobs are reliably executed on computing resources that satisfy the explicit requirements of the routine.

Features

Condor provides many features that are necessary for a distributed continuous integration system like the NMI framework [24]. It would be possible to deploy the framework using a different batch system if the system implemented capabilities similar to the following found in Condor.

Matchmaking: Condor uses a central negotiator for planning and scheduling jobs for execution in a pool. Each machine provides the negotiator with a list of its capabilities, system properties, pre-installed software, and current activity. Jobs waiting for execution also advertise their requirements that correspond to the information provided by the machines. After Condor collects this information from both parties, the negotiator pairs jobs with resources that mutually satisfy each other's requirements. The matched job and resource communicate directly with each other to negotiate further terms, and then the job is transferred by Condor to the machine for execution. The framework will warn users if they submit a build or test with a requirement that cannot be satisfied by any machine in the pool.

Fault tolerance: The failure of a single component in a Condor pool only affects those processes that deal directly with it. If a computing resource crashes while executing a build-and-test routine, Condor can either migrate the job to another machine or restart it when the resource returns. Condor uses a transient lease mechanism to ensure only a single instance of a job exists in a pool at any one time. If a computing resource is unable to communicate with the central negotiator when a job finishes execution, Condor transfers back the retained results once network connectivity is restored.

Grid resource access: Condor enables users to access computing resources in other pools outside of their local domain. Condor can submit jobs to grid resource middleware systems to allow builds and tests to execute on remote machines that may or may not be running Condor [11].

Resource control A long-standing philosophy of the Condor system is that the resource owner must always be in control of their resource, and set the terms of its use. Owners that are inconvenienced

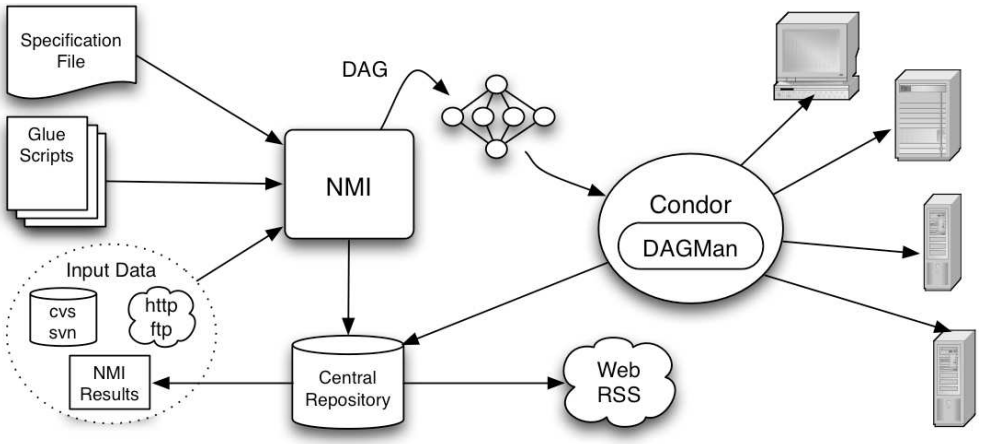

Figure 3: NMI Framework Architecture – The user submits a new routine comprised of glue scripts, input data, and a workflow specification file. The NMI software uses this information to create a dependency execution graph and submits a DAGMan job to the Condor batch system. When the DAGMan job begins to execute, it deploys multiple Condor jobs to the build-and-test computing resources. All output data produced by the routine's jobs are stored in a central repository and retrieved through ancillary clients.

by sharing their resources are less likely to continue participation in a distributed build-and-test pool. Condor provides flexible policy expressions that allow administrators to control which users can access resources, set preferences for certain routines over others, and limit when users are allowed to execute builds and tests.

Authentication: Condor supports several authentication methods for controlling access to remote computing resources, including GSI [9], Kerberos [23], and Microsoft's SSPI [1].

File transfer The NMI framework uses Condor's built-in file transfer protocol to send data between submission hosts and build-and-test resources. This robust mechanism ensures that files are reliably transferred; transfers are automatically restarted upon connection failure or file corruption. Condor can also use a number of encryption methods to securely transfer files without a shared file system.

Pool Configuration

Condor is designed to balance the needs and interests of resource owners, users wanting to execute jobs, and system administrators. In this spirit, Condor enables administrators to deploy and manage build-and-test pools that respect the wishes of resource owners but can still provide access for users. Priority schemes for both dedicated and non-dedicated resources can be created using Condor's flexible resource policy expressions. For example, the dedicated resources in a pool may prefer to execute processor-intensive builds and high-load stress tests so that shorter tests can be scheduled on idle workstations. Preferential job priority may also be granted to specific users and groups at different times based on deadlines and release schedules.

Condor can also further divide the resources of individual build-and-test machines, similar to the policies for the entire pool. Condor can allocate a multiprocessor machine's resources disproportionately for each processor. For example, in one configuration a processor can be dedicated for build routines and therefore is allocated a larger portion of the system's memory. Test routines are only allowed to execute on the processor with more memory when no other jobs are waiting for execution. If a build is submitted while a test job is executing on this processor, Condor automatically evicts the test job and restarts it at a later time.

Build-and-test pools often have periods where there are no new routines available for execution. If a computing resource is idle for certain length of time, Condor can trigger a special task in the framework that performs continuous tests against an application as *backfill*. This is useful to perform long-term stress and random input tests on an application [18]. The results from these tests are reported back to the central repository periodically or whenever Condor evicts the backfill job to run a regular build or test routine.

Pool & Resources Management

We now discuss our experiences in managing the NMI Build & Test laboratory at the University of Wisconsin-Madison. The NMI framework is also currently deployed and running in production at other locations, including multi-national corporations and other academic institutions.

Operating System	Versions	Archs	CPUs
Debian Linux	1	1	2
Fedora Core Linux	4	2	20
FreeBSD	1	1	4
HP HPUX	1	1	3
IBM AIX	2	1	6
Linux (Other)	3	2	9
Macintosh OS X	2	2	8
Microsoft Windows	1	2	3
OSF1	1	1	2
Red Hat Linux	3	2	13
Red Hat Enterprise Linux	2	3	19
Scientific Linux	3	2	11
SGI Irix	1	1	4
Sun Solaris	2	1	6
SuSE Enterprise Linux	3	3	15

Table 1: NMI Build & Test Laboratory Hardware – The laboratory supports multiple versions of operating systems on a wide variety of processor architectures.

Our facility currently maintains over 60 machines running a variety of operating systems (see Table 1). Over a dozen projects, representing many developers and institutions, use the NMI laboratory for building and testing grid and distributed computing software. In order to fully support the scientific community, we maintain multiple versions of operating systems on different architectures. Machines are not merely upgraded as newer versions of our supported platforms are released. We must instead install new hardware and maintain support for older platform combinations for as long they are needed by users.

Resource Configuration

We automate all persistent software installations and system configuration changes on every machine in our build-and-test pool. Anything that must be installed, configured, or changed after the default vendor installation of the operating system is completely scripted, and then performed using cfengine [4]. This includes installing vendor patches and updates. Thus, new machine installations can be added to the facility without requiring staff to rediscover or repeat modifications that were made to previous instances of the platform.

Prerequisite Software

In a multi-user build-and-test environment, projects often require overlapping sets of external software and libraries for compilation and testing. The NMI framework lets administrators offer prerequisite software for

routines in two ways: (1) the external software can be pre-installed on each computing resource and published to the NMI system, or (2) the system can maintain a cache of pre-compiled software to be deployed dynamically when requested by a user. Dynamic deployment is advantageous in environments where routines may execute on resources outside of one administrative domain and are unable to expect predictable prerequisite software.

At the NMI Laboratory, we use cfengine to install a large set of prerequisite software on each of our computing resources. This eases the burden on new users whose builds expect a precise set of non-standard tools but are not prepared to bring them along themselves. The trade-off, however, is that these builds and tests are less portable across administrative domains.

Data Management

The NMI Laboratory produces approximately 150 GB of data per day. To help manage the large amount of data generated by builds and tests, the framework provides tools and options for administrators.

Multiple submission points: More than one machine can be deployed as a submission host in a build-and-test pool. By default, the output of a routine is archived on the machine it is submitted from. The framework provides a built-in mechanism to make these files accessible from any submission host without requiring users to know which machine the data resides on. If a user requests output files from a previous build on a different submission host, the framework automatically transfers the files from the correct location.

Repository pruning The framework provides mechanisms for removing older build and test results from the repository based on flexible policies defined by the lab administrator. When the framework is installed on a submission host it deploys a special job into the batch system that periodically removes files based on the administrator's policy. Routines may be pruned based on file size, submission date, or other more complicated properties, such as duplicate failures. This process will only remove user-specified results; task output log files, error log files, and input data are retained so that builds and tests are reproducible. Users can set a routine's preservation time stamp to prevent their files from being removed before a certain date.

Case Studies

The NMI Laboratory is used as a build and test facility for two large distributed computing research projects: the Globus Toolkit from the Globus Alliance [8], and the Condor batch system from the University of Wisconsin-Madison's Department of Computer Sciences [15]. We present two brief case studies on how the NMI framework has improved each of these projects software development process.

Globus Toolkit

The Globus Toolkit is an open source software distribution that provides components for constructing large grid systems and applications [8]. It enables users to share computing power and other resources across administrative boundaries without sacrificing local autonomy. Globus-based systems are deployed all across the world and are the backbone of many large academic projects and collaborations.

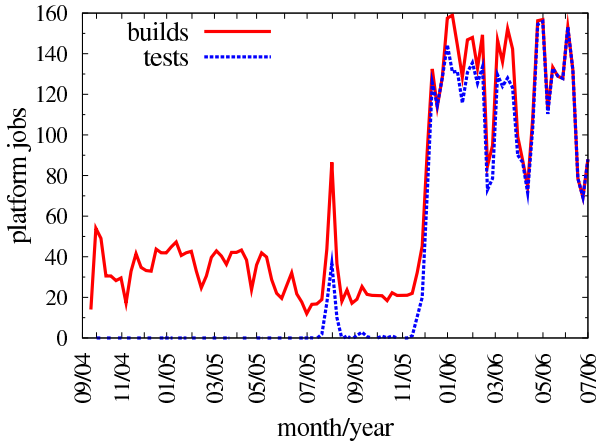

Figure 4: Globus Builds & Tests – The large spike in the number of jobs in the graph indicates when a new version of Globus was released and required many new build and test routines. Initially, the toolkit's build-and-test procedures were contained in a monolithic batch script. The tests were then later broken out of the build scripts into separate tasks. Thus, no data exists on these tests that were executed in the first months after switching to the NMI system.

Prior to switching to the NMI framework, the Globus system was built and tested using a combination of custom scripts and the Tinderbox open-source continuous integration system [20]. Each build machine contained a pre-defined source file that mapped all the external software needed by the build process to paths on the local disk. This file contained the only record in the system of what external software was used to execute a build or test, and did not contain full information about the specific version used. If the computing resource was updated to use a newer version of the software, there was no record in the build system to reflect that fact.

As the project grew, developers received an increased amount of bug reports from users. Many of these reports were for esoteric platforms that were not readily available to the Globus developers. Fewer builds and tests were submitted to these machines, which in turn caused bugs and errors to be discovered much later after they were introduced into the source code.

Now the Globus Toolkit is built and thoroughly tested every night by the NMI Build & Test software

on 10 different platforms (Figure 4). The component glue scripts for Globus contain the same build procedures that an end-user follows in order to compile the toolkit. These procedures also include integrity checks that warn developers when the build process generates files that are different from what the system expected. All other regression and unit tests are preformed immediately after compilation. Globus' developers have benefited from the NMI framework's strict attention to the set of software installed on computing resources and its ability to maintain a consistent execution environment for each build-and-test run. This allows them to test backwards compatibility of their build procedures with older versions of development tools, which they were unable to do before.

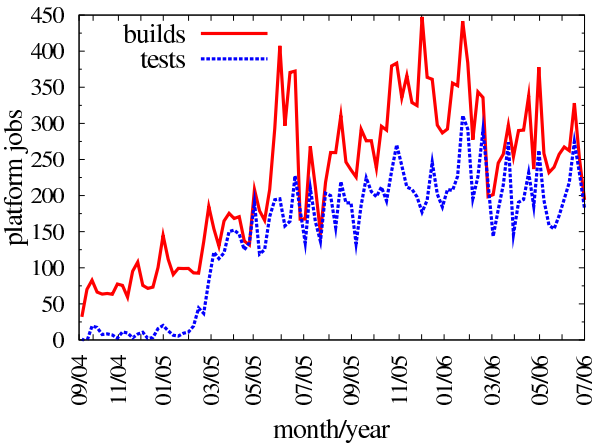

Figure 5: Condor Builds & Tests – Each platform job is a single build or test execution cycle on a computing resource; there may be multiple platform jobs for a single framework build-and-test routine. Sharp increases in the number of builds correspond to release deadlines for developers.

Condor

Before the advent of Linux's popularity, Condor supported a modest number of operating systems used by the academic and corporate communities. Initially, each developer was assigned a platform to manually execute builds and given a paper checklist of tests to perform whenever a new production release was needed. All of Condor's build scripts contained hard-coded path information for each machine that it was built on. If one of these machines needed to be rebuilt or replaced, the administrator would have to construct the system to exactly match the expected specification.

Like Globus, the Condor development team also deployed a Tinderbox system to automate builds and tests on all the platforms that were supported. Due to hardware and storage limitations, however, this system could only build either the stable branch or the development branch of Condor each day; developers had to make a decision on which branch the system should build next. This also meant that the system could not easily build custom branches or on-demand builds of developer's workspaces.

Since transitioning to the NMI framework, the Condor project has experienced a steady increase in the number of builds and tests (Figure 5). The development team submits an automatic build and test to the framework every night for both the stable and development releases; Condor is built on 17 platforms with 122 unit and regression tests per platform. In addition, the framework is used for numerous on-demand builds of Condor submitted by individual developers and researchers to test and debug experimental features and new platforms.

Future Work

Many facets of the NMI framework can be expanded to further improve its capabilities.

Currently, the NMI framework coordinates builds and tests on multiple platforms independently. Each routine executes on a single computing resource for each specified platform. We are developing a mechanism whereby a build-and-test routine can execute on multiple machines in parallel and allow them to communicate with one another. Users specify an arbitrary number of machines and the batch system deploys the routine only when it has simultaneous access to all of the resources it requires. The framework passes information to the glue scripts about which machines are running the other parallel instances of the routine. Such dynamic cross-machine testing will allow users to easily test platform and version interoperability without maintaining permanent "target" machines for testing.

We are also extending our test network into the Schooner [21] system, based on Emulab [26], to expand these distributed tests to cover a variety of network scenarios. Schooner permits users to perform tests which include explicit network configurations. For example, the NMI framework will be able to include automated tests of how a distributed application performs in the presence of loss or delay in the network. This system will also allow administrators to rapidly deploy a variety of different operating system configurations both on bare hardware and in virtual machines.

A major boon to the NMI framework will be the proliferation of virtualization technology in more systems. Instead of deploying and maintaining a specific computing resource for every supported platform, the framework would keep a cache of virtual machine images that would be dynamically deployed at a user's request. Because administrators will only need to configure a single virtual machine image for each operating system in the entire pool, this will simplify build-and-test pool management and utilization. The framework would then also be able to support application testing that requires privileged system access or which makes irreversible alterations to the system configuration; these

changes would be localized to that instance of the virtual operating system and not the cached image.

Availability

The NMI Build & Test Laboratory continuous integration framework is available for download at our website under a BSD-like license: http://nmi.cs.wisc.edu/ .

Acknowledgments

This research is supported in part by NSF Grants No. ANI-0330634, No. ANI-0330685, and No. ANI-0330670.

Conclusion

We have presented the NMI Build & Test Laboratory continuous integration framework software. Our implementation is predicated on design principles that we have established for distributed build-and-test systems. The key features that distinguish our system are (1) its ability to execute builds and tests on computing resources spanning administrative boundaries, (2) it is deployed dynamically on heterogeneous resources, and (3) it maintains a balance between continuous integration practices and on-demand access to builds and tests. Our software uses the Condor batch system to provide the capabilities necessary to operate in a distributed computing environment. We discussed our experiences in managing a diverse, heterogeneous build-and-test facility and showed how the NMI framework functions as the primary build-and-test system for two large software projects. From this, we believe that our system can be used to improve the development process of software in a distributed computing environment.

Author Biographies

Andrew Pavlo, Peter Couvares, Rebekah Gietzel, and Anatoly Karp are members of the Condor research project at the University of Wisconsin-Madison's Department of Computer Sciences. Ian D. Alderman is a Ph.D. candidate at the University of Wisconsin-Madison's Department of Computer Sciences. Miron Livny is a Professor with the Department of Computer Sciences at the University of Wisconsin-Madison and currently leads the Condor research project.

Charles Bacon is a researcher specializing in grid technology at Argonne National Laboratory.

Bibliography

[1] *The security support provider interface,* White paper, Microsoft Corp., Redmond, WA, 1999.

[2] Beck, K., *Extreme programming explained: embrace change*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[3] Boehm, B. W. and P. N. Papaccio, "Understanding and controlling software costs," *IEEE Transactions Software Engineering 14*, Vol. 10, pp. 1462-1477, 1988.

[4] Burgess, M., "A site configuration engine," *USENIX Computing Systems*, Vol. 8, Num. 2, pp. 309-337, 1995.

[5] Couvares, P., T. Kosar, A. Roy, J. Weber, and K. Wenger, *Workflows for e-Science*, Chapter: Workflow Management in Condor, Springer-Verlag, 2006.

[6] *CruiseControl*, http://cruisecontrol.sourceforge.net .

[7] Fierro, D., *Process automation solutions for software development: The BuildForge solution*, White paper, BuildForge, Inc., Austin, TX, March, 2006.

[8] Foster, I., and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, Vol. 11, Num. 2, pp. 115-128, Summer, 1997.

[9] Foster, I. T., C. Kesselman, G. Tsudik, and S. Tuecke, "A security architecture for computational grids," *ACM Conference on Computer and Communications Security,*, pp. 83-92, 1998.

[10] Fowler, M., *Continuous integration*, May, 2006, http://www.martinfowler.com/articles/continuous Integration.html .

[11] Frey, J., T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," *Cluster Computing*, Vol. 5, pp. 237-246, 2002.

[12] Grenning, J., "Launching extreme programming at a process-intensive company," *IEEE Software*, Vol. 18, Num. 6, pp. 27-33, 2001.

[13] Hellesoy, A., *Continuous integration server fea ture matrix*, May, 2006, http://damagecontrol.codehaus.org/Continuous+Integration+Server+Feature+Matrix .

[14] Holck, J., and N. Jørgensen, "Continuous integration and quality assurance: A case study of two open source projects," *Australian Journal of Information Systems*, Num. 11/12, pp. 40-53, 2004.

[15] Litzkow, M., M. Livny, and M. Mutka, "Condor – a hunter of idle workstations," *Proceedings of the 8th International Conference of Distributed Computing Systems*, June, 1988.

[16] *Apache Maven*, http://maven.apache.org .

[17] Mcconnell, S., "Daily build and smoke test," *IEEE Software*, Vol. 13, Num. 4, p. 144, 1996.

[18] Miller, B. P., L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the Association for Computing Machinery*, Vol. 33, Num. 12, pp. 32-44, 1990.

[19] Ousterhout, J., and J. Graham-Cumming, *Scalable software build accelerator: Faster, more accurate builds*, White paper, Electric Cloud, Inc., Mountain View, CA, February, 2006.

[20] Reis, C. R., and R. P. de Mattos Fortes, "An overview of the software engineering process

and tools in the Mozilla Project," *Workshop on Open Source Software Development*, Newcastle, UK, pp. 162-182, 2002.

[21] *Schooner*, http://www.schooner.wail.wisc.edu .

[22] Schuh, P., "Recovery, redemption, and extreme programming," *IEEE Software*, Vol. 18, Num. 6, pp. 34-41, 2001.

[23] Steiner, J. G., B. C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," *Proceedings of the USENIX Winter 1988 Technical Conference*, USENIX Association Berkeley, CA, pp. 191-202, 1988.

[24] Tannenbaum, T., D. Wright, K. Miller, and M. Livny, "Condor – a distributed job scheduler," *Beowulf Cluster Computing with Linux*, T. Sterling, Ed., MIT Press, October, 2001.

[25] Thain, D., T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience," *Concurrency – Practice and Experience*, Vol. 17, Num. 2-4, pp. 323-356, 2005.

[26] White, B., J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, pp. 255-270, USENIX Association, Dec., 2002.